ARH 6325

Advanced Communications Systems Development

Coding Conventions

July 23, 1984

This document describes the conventions and procedures to be used by programmers in the development and maintenance of Distributed Communications Network Software (DCNS) in the CYBIL implementation language.

Table of Contents

1.0 INTRODUCTION

1.1 PURPOSE

This document describes the conventions and procedures to be used by programmers in the development and maintenance of Distributed Communications Network Software (DCNS) in the CYBIL implementation language.

There are a variety of routine, mundane aspects associated with writing programs. A set of coding conventions remove from the programmer trivial decisions relating to module format, name generation, etc. thereby leaving more time to concentrate on the important matters.

During the lifetime of a large software product, the average developer will come in contact with a large number of programs written by and maintained by many other programmers. A consistent set of coding conventions helps the programmer "feel at home" with a new program and therefore is able to begin doing useful work sooner.

The implementation of these conventions will increase the efficiency of program development, improve the reliability and maintainability of the program and aid in the training of persons who will be maintaining or using the program.

1.2 REFERENCE DOCUMENTS

Programmers using this convention should also be familiar with the CYBIL Implementation Dependent Handbook (DCS #ARH3078). This convention used the "CYBIL Program Library Conventions" and the "CYBIL Coding Conventions" sections from the CYBIL Handbook as guidelines. The following sections from the Handbook are also of particular interest:

- "CYBIL-CM/IM Type and Variable Mapping" - describes the MC68000 data formats for each of the supported CYBIL data types.

- "CYBIL-CM/IM Run Time Environment" - describes the run time environment: memory, parameter passing, variables, heap management, etc.

- "Procedure Interface Conventions" - describes conventions that should generally be used by designers of procedural interfaces. The conventions in this section should be followed by programmers using the ACSD Coding Convention.

- "Efficiencies (General and CM)" - lists a group of programming tips to help the user better utilize the CYBIL development environment. It will be added to as additional tips become known.

- "Implementation Limitations (General and CM)" - describes implementation limitations.

The ERS for the CYBIL Formatter V1.0 (DCS #ARH2619) is another document that programmers using this convention should be familiar with. This document describes CYBFORM, the utility used to format CYBIL source code to maximize readability.

CYBFORM is the major software tool for enforcing ACSD coding conventions. All programs MUST be run through the formatter. If additional code is added after this process, it is required to be in the same format or the program must be run through CYBFORM again. The formatter should be responsible for:

- all indentation
- capitalizing keywords
- inserting the proper number of blank lines
- supporting 95 character source lines
- repeating the label at the end of a structured block
- etc.

1.3 CONFORMANCE AND ENFORCEMENT

The spirit of the conventions as well as the specifics should be adhered to. The enforcement of these conventions and the acceptance of "reasonable deviations" are the responsibility of the code reviewers. Programs which do not conform to these conventions will be returned to the programmer for correction.

If parts of the convention are found to be unworkable or if some issues are found not to be covered, this document should be updated.

2.0 DOCUMENTATION

2.1 PURPOSE

The primary purpose of documentation is to help someone other than the original developer understand what the code is doing.

2.2 GENERAL REQUIREMENTS

Comments within CYBIL code should provide nonredundant information describing why or what a series of CYBIL statements are doing.

All documentation and comment lines should contain complete English sentences with correct punctuation. Exceptions are allowed in embedded comments, titles and data declaration stand-alone comments (such as "Work codes for intertask messages.").

Documentation should avoid the use of personal pronouns (he, she, him, her, etc.).

When feasible, comments should be kept general enough so that they will not become outdated by detailed changes to the code. Where values are defined by constants (CONST), the name rather than the value should be referenced in the documentation.

The abbreviations for technical terms which are to be used in documentation are listed in Appendix A. A module whose documentation makes extensive use of terms not in this list may define a list of abbreviations and include it in the module level documentation. All other technical words and phrases must be completely spelled out. Routine names and mnemonic names of tables and hardware components are not considered abbreviations.

2.3 PROLOGUE DOCUMENTATION

The information contained in a prologue should be at the level indicated by the complexity of the code which it is documenting.

Defined formats exist for the information provided in prologue documentation. The general format is as follows:

2.0 DOCUMENTATION
2.3 PROLOGUE DOCUMENTATION

- An uppercase keyword appears in column 3 followed by a ":".

- Text begins on the next line indented two spaces from the keyword, and may continue (at the same level of indentation) for as many lines as necessary.

- Blank comment lines may be used to separate paragraphs, but the indentation level should be maintained.

- Where lists of variables, parameters, acronyms or abbreviations and a description of each are used, the description continuation lines (if necessary) are indented 6 columns from the keyword.

- Each keyword and the associated text must be separated from the next keyword by a blank comment line.

The keywords that apply to each of the prologue documentation types are defined in the following sections.

2.3.1 MODULE PROLOGUES

```
{ PURPOSE:
{   A short description of what the module does.  This should
{   contain the purpose of the module and the reasons for
{   grouping the declarations in the module, rather than
{   describing the purpose of each procedure.
{
{ DESIGN:
{   A one or two paragraph overview description of the design
{   of the module.  This should describe how the module works
{   in general terms.  Usage of specific variable or procedure
{   names is discouraged.
{
{ GLOSSARY:
{   A list of acronyms and abbreviations used in this module
{   (which are not defined in the appendix to this convention)
{   and their meaning.  This section may be omitted if no new
{   acronyms or abbreviations are used in this module.
{
{ Copyright Control Data Corporation, 1984.
```

2.3.2 PROGRAM/PROCEDURE/FUNCTION PROLOGUES

For programs, procedures and functions, the first line of the prologue block must contain the routine type followed by the routine name:

```
{ PROGRAM program_name          or

{ PROCEDURE procedure_name      or

{ FUNCTION function_name
```

This line is followed by a blank comment line and then the list of keywords and the associated text appropriate for the routine. The keywords which may be used are listed below. Any keywords which are not applicable are to be omitted. The keywords which are used should appear in the order in which they are described below.

- PURPOSE
- DESCRIPTION
- CALL FORMAT
- COMMAND FORMAT
- ENTRY CONDITIONS
- EXIT CONDITIONS
- ERROR CONDITIONS
- INTERTASK MESSAGE INPUT
- INTERTASK MESSAGE OUTPUT
- GLOBAL DATA REFERENCED
- GLOBAL DATA MODIFIED
- NOTES AND CAUTIONS

This format is used for XREF/XDCL procedure and function header common decks, for inline procedure common decks, and for procedures and functions within a module.

2.3.2.1 PURPOSE

This section contains a short description of the process the procedure or function performs (rather than the method used). This section is required.

Example:

```
{ PURPOSE:
{   Perform a physical copy of a message to a new buffer chain.
```

2.3.2.2 DESCRIPTION

This section contains a one or two paragraph description of how the procedure works. Normally this section should be included, although for some short routines the PURPOSE section may be sufficient.

Example:

```
{ DESCRIPTION:
{   A message is physically copied to new buffers, and the old
{   set of buffers is released.  Data is compact in the new
{   buffers; the first (n-1) buffers are full, and the last one
{   has all of its empty space in the trailing portion of the
{   buffer.
```

2.3.2.3 CALL_FORMAT

This section is required for common inline and externally callable routines. A line showing the calling format followed by a description of each of the parameters should be included.

Example:

```
{ CALL FORMAT:
{   (*calic clxgcs)
{   CLP_GET_SET_COUNT (PARAMETER_NAME, PVT, VALUE_SET_COUNT,
{         STATUS)
{
{   PARAMETER_NAME: (input)  This parameter specifies the name
{         of the parameter for which the value set count is to
{         be returned.  Any one of the names for the parameter
{         may be specified.
{
{   PVT: (input)  This parameter specifies the Parameter Value
{         Table for the parameter list.
{
{   VALUE_SET_COUNT: (output)  The number of value sets given
{         for the specified parameter is returned in this
{         parameter.
{
{   STATUS: (output)  The status of the request is returned in
{         this parameter.
```

2.3.2.4 COMMAND_FORMAT

This section is required for command processors. It is used to describe the command (rather than the CALL FORMAT section used to describe the procedure interface). The first line of text should contain the singular form of the command name, plural form or alternate spelling of the command name (if any), and the abbreviation in the following format:

```
{ COMMAND NAME or PLURAL COMMAND NAME (ABBREVIATION)
```

Following this line, each of the parameters should be described. The parameter name and alternate forms of the name should be specified followed by the allowable values for the parameter and a description of the parameter. The description should include the default value for the parameter (if any).

Example:

```
{ COMMAND FORMAT:
{   SET_TIME (SETT)
{
{   HOUR (HOURS or H): 0..23  This parameter specifies the
{         hour to set in the system clock.
{
{   MINUTE (MINUTES or M): 0..59  This parameter specifies
{         the minutes to set in the system clock.
{
{   SECOND (SECONDS or S): 0..59  This parameter specifies
{         the seconds to set in the system clock.
```

2.3.2.5 ENTRY_CONDITIONS

This section describes any conditions which must be met before the routine is called. Information concerning parameters may be specified if the comments on the program/procedure/function declaration are not sufficient. Entry conditions also include such things as the logical status of connections, files, buffers, etc.

Example:

```
{ ENTRY CONDITIONS:
{   The user_fcb record must be initialized via an open_file
{   request to the file_access procedure.
```

2.3.2.6 EXIT_CONDITIONS

This section describes any conditions which exist on a normal return from the routine, which the caller should be aware of. A description of the return variable for functions (and procedures) may be specified if the comment on the function (or procedure) declaration is not sufficient. Exit conditions also include such things as the logical status of connections, files, buffers, etc.

Example:

```
{ EXIT CONDITIONS:
{   If the specified key is found in the tree, the associated
{   data entry is raturned, otherwise the procedure return value
{   will be NIL.
```

2.3.2.7 ERROR_CONDITIONS

This section describes any error conditions which exist on a return from the routine, which the caller should be aware of. Special processing done or parameter values returned are described. Error conditions may also include such things as the logical status of connections, files, buffers, etc.

Example:

```
{ ERROR CONDITIONS:
{   The value of parameters line_number and command will be
{   undefined if an access error occurs when reading the file.
```

2.3.2.8 INTERTASK_MESSAGE_INPUT

This section contains a list of intertask messages received by this procedure. The intertask message workcode should be listed, followed by e description of the message.

Example:

```
{ INTERTASK MESSAGE INPUT:
{   exec_tskfail - This intertask message indicates that a task
{         has failed due to a bus error or an address error.
{   sa_start_task_for_user - This intertask message requests that
{         a task be started for a user task such that System
{         Ancestor is the parent.
```

2.3.2.9 INTERTASK_MESSAGE_OUTPUT

This section contains a list of intertask messages sent by this procedure. The intertask message workcode should be listed, followed by a description of the message.

Example:

```
{ INTERTASK MESSAGE OUTPUT:
{   mci_regulation_change - This intertask message is used to tell
{          the SSR that a new regulation level is now to be put into
{          effect.
```

2.3.2.10 GLOBAL_DATA_REFERENCED

This section contains a list of the global data referenced.

Example:

```
{ GLOBAL DATA REFERENCED:
{   sys_cnfg.running
{   sys_cnfg.binclock
```

2.3.2.11 GLOBAL_DATA_MODIFIED

This section contains a list of the global data modified and a description of the modification.

Example:

```
{ GLOBAL DATA MODIFIED:
{   command_source - the address of the user that issued the
{          command is saved.
```

2.3.2.12 NOTES_AND_CAUTIONS

This section documents design, implementation and general information which may be useful to other analysts, especially any uncommon, unusual, or obscure techniques used by the coder. Also include warnings about problems that might be encountered during modification of this routine.

Example:

```
{ NOTES AND CAUTIONS:
{   This is a highly time consuming operation, requiring at
{   least 3-5 microseconds per byte copied.  It is recommended
{   that the caller either run at a relatively low task
{   priority, or yield control sometime after the routine
{   returns to avoid time slice overrun, and to permit other
{   processes to be active.
```

2.3.3 DECLARATION (CONST, TYPE AND VAR) PROLOGUES

Prologues for groups of declarations contain the keyword "DESCRIPTION" . The keyword is followed by one or more paragraphs describing what the relationship between the declarations is (what they are for, how they are used and/or why they are grouped together).

This format is used for constant and type ,declaration common decks, header decks for XREF variable declaration common decks, and may also precede blocks of related module level declarations. Common decks always contain a one line description and a blank comment line, which precedes the prologue. For example:

```
{ CMDTTRE - Tree Management Definitions.
{
```

2.0 DOCUMENTATION
2.3.3 DECLARATION (CONST, TYPE AND VAR) PROLOGUES

The prologue for declarations within a module should contain a similar short description line before the DESCRIPTION keyword and text. The declarations may be preceded by a page eject and a title. For example:

```
?? NEWTITLE := 'Directory M-E Data Stores', EJECT ??
{ Directory M-E Data Stores.
{
{ DESCRIPTION:
{   The following declarations define the data stores maintained
{   by the Directory M-E.  These are the Registered Data Store
{   (RDS), the Translation Data Store (TDS), and the Translation
{   Request Data Store (TRDS).
{
{   The RDS contains all the currently registered titles.
{   These titles were created by a primitive.
{
{   The TDS contains a list of the most recent translations
{   received from other systems.  The least recently used entry
{   is deleted.  A threshold number of entries are held in this
{   system.
{
{   The TRDS contains all the currently active translation
{   requests.
```

2.4 CODE LEVEL DOCUMENTATION

Source code is the ultimate documentation of any program. Therefore, in all ACSD programming, a consistent emphasis should be placed on producing lucid, readable and self-documenting code.

Comments in the source code should only provide information not readily apparent from reading the code. If descriptive procedure, parameter and variable names are used, the number of inline comments should be minimal.

In addition to prologue documentation, there are two types of code level comments: embedded and stand alone comments.

Embedded comments appear on the same line following a declaration or executable statement. The comment need not be a complete sentence. This type of comment should not be continued onto another line. If the intended comment is too long to fit on the single line, it should be inserted as a stand alone comment preceding the area of code to which it applies. Embedded comments should be used to convey software or system attributes which are not discernable from CYBIL declarations.

2.0 DOCUMENTATION
2.4 CODE LEVEL DOCUMENTATION

Stand alone comments are blocks of text appearing inline with code, as opposed to within the prologue blocks previously defined. All stand alone comments are preceded and followed by one blank line. The comments are complete English sentences with correct punctuation.

3.0 NAMING CONVENTIONS

3.0 NAMING CONVENTIONS

3.1 GENERAL

The key to making programs readable is the usage of meaningful, noncryptic English names for all CYBIL constructs. Avoid the use of special characters ("$", "@", and "#") in local names since these characters may have special meaning for global system naming conventions.

Names should be chosen for how they will read in the code body of a routine, not how they look in the data declaration. This is especially true of variables and field names in TYPE declarations.

For example:

```
TYPE
  program_descriptor = record
    load_map: load_map_options,
  recend,

  load_map_options = record
    file_name: file_name,
    options: (all,nothing),
  recend;

VAR
  my_program: program_descriptor;

  ...

my_program.load_map.file_name := 'LOADMAP';
```

Procedure and functions names should describe the process the procedure performs.

CYBIL statement labels are often needed in order to perform EXIT or CYCLE commands. Sometimes, structurally unnecessary labels can be powerful documentary aids and their use is encouraged. Label names should always describe the function being performed by the structured statement to which they refer.

3.0 NAMING CONVENTIONS
3.1 GENERAL

For example:

```
/search_symbol_table/     (instead of)

/label1/
```

Boolean names should always describe the TRUE condition.

For example:

```
IF file_is_open THEN     (instead of)

IF file_switch THEN
```

3.2 DECK NAMING CONVENTIONS

Deck names have the following format:

ggtxxxx - where

gg = two character group name.
t = one character indicating deck type.
xxxx = one to four character name/mnemonic for uniqueness within the group.

Allowable codes for the deck type are as follows:

A = MC68000 Assembler code
B = Binder directives (common deck)
C = Common deck calls (common deck)
D = CYBIL Constant and type declarations (common deck)
F = Installation procedure directives (common deck)
H = Documentation header (common deck)
I = CYBIL inline procedure (common deck)
L = Linker Directives (common deck)
M = CYBIL code module
T = Type identifiers (common deck)
X = CYBIL XREF declaration (common deck)
7 = Cyber 170 test stubs
8 = Cyber 180 test stubs

Decks of type M must consist of exactly one module (compilation unit).

Each group has a history deck named gg (where "gg" is the group name). This deck contains a history of all decks in the group which were added to the PL, deleted from the PL or modified.

4.0 CODE LAYOUT

4.0 CODE_LAYOUT

The following sections define the code layout for modules, programs, procedures, functions and common decks.

In general, declarations should be listed alphabetically within each declaration grouping (CONST, TYPE and VAR). Where a logical order is more meaningful, then that grouping may be used and a comment should describe what the logical grouping is.

A logical order would be appropriate for related declarations such as the type and variable declarations for a table, or a major type and it's subordinate types.

4.1 LAYOUT_CONTROL

4.1.1 PAGE EJECTS

Page ejects inserted in appropriate places can make source listings more readable. All programs, procedures, functions and other large items should be preceded with EJECT calls.

4.1.2 TITLES

Titles are required for each module, program, procedure, function and common deck. In addition, their use throughout the module is recommended to indicate such things as global variables and functional groupings of procedures.

For example, the following title directives may be some of those used for a module:

```
?? TITLE := 'COCNET: Directory Management Entity' ??

?? NEWTITLE := 'Directory M-E Data Stores', EJECT ??
?? OLDTITLE ??

?? NEWTITLE := 'PROGRAM dir_init', EJECT ??
?? OLDTITLE ??

?? NEWTITLE := 'Directory Registration Procedures' ??
?? NEWTITLE := 'PROCEDURE [#GATE,XDCL] dir_change', EJECT ??
?? OLDTITLE ??
```

4.0 CODE LAYOUT
4.1.2 TITLES

```
?? NEWTITLE := 'PROCEDURE [#GATE,XDCL] dir_create', EJECT ??
?? OLDTITLE ??
?? NEWTITLE := 'PROCEDURE [#GATE,XDCL] dir_delete', EJECT ??
?? OLDTITLE ??
?? NEWTITLE := 'PROCEDURE [INLINE] dir_rcb_init', EJECT ??
?? OLDTITLE ??
?? OLDTITLE ??

?? NEWTITLE := 'Directory Translation Procedures' ??
?? NEWTITLE := 'PROCEDURE [#GATE,XDCL] dir_abort', EJECT ??
?? OLDTITLE ??
?? NEWTITLE := 'PROCEDURE [INLINE] dir_tcb_init', EJECT ??
?? OLDTITLE ??
?? NEWTITLE := 'PROCEDURE [#GATE,XDCL] dir_translate', EJECT ??
?? OLDTITLE ??
?? OLDTITLE ??
```

Note that the use of NEWTITLE and OLDTITLE allows "stacked" listing titles to exist.

4.1.3 LISTING DIRECTIVES

The LISTEXT listing toggle must be used to control the listing of common decks. The calling deck may use the "?? PUSH (LISTEXT := ON) ?? / ?? POP ??" directives around some or all common decks to allow the listing to be controlled by the "LO" list option on the CYBIL compiler command. The Module, Procedure and Common deck layout sections describe the directives to use and where they are to be located in the decks.

4.0 CODE LAYOUT
4.2 MODULE LAYOUT

4.2 MODULE LAYOUT

The code layout for a module is as follows:

```
ggMxxxx
*ifcall,BUILD,ggTxxxx
?? TITLE := 'CDCNET: module purpose' ??
MODULE <module_name>;

{ Module prologue
{ .
{ .
{ .
{ (continue module prologue)

?? LEFT := 1, RIGHT := 95 ??
?? FMT (FORMAT := ON, KEYW := UPPER, IDENT := LOWER) ??

?? PUSH (LISTEXT := ON) ??
<global *calls of common decks>
?? POP ??

<global TYPE, CONST and VAR declarations>

<program/procedure/function declarations>

MODEND <module_name>;
```

The LISTEXT toggle is generally used to control listing of called common decks, however, common decks of major importance to the module may always be listed. For example:

```
<global *calls of common decks of major importance>
?? PUSH (LISTEXT := ON) ??
<global *calls of rest of common decks>
?? POP ??
```

4.3 PROGRAM/PROCEDURE/FUNCTION LAYOUT

All CYBIL routines (including nested procedures) should begin on a new listing page and start with a prologue. A NEWTITLE line (which includes the page eject) precedes the prologue and specifies the type of routine ("PROGRAM", "PROCEDURE" or "FUNCTION"), attributes (if any) and the routine name.

An OLDTITLE follows the PROCEND or FUNCEND statement.

4.0 CODE LAYOUT
4.3 PROGRAM/PROCEDURE/FUNCTION LAYOUT

In declarations of parameter lists, always separate formal parameters by coding each parameter on a separate line. The program/procedure/function declaration may include comments following each parameter stating whether it is input (I), output (O), or both (I/O), followed by a description of the parameter.

Always declare all input parameters before all output parameters unless there is an obvious symmetry that would be disturbed.

The routine layout is as follows:

```
?? NEWTITLE := 'PROCEDURE [attributes] proc_name', EJECT ??

{ Procedure prologue
{ .
{ .
{ .


PROCEDURE proc_name ( {
      par_1: type; { I: description of parameter 1
  VAR par_2: type; { I/O: description of parameter 2
      .              .
      .              .
      .              .
  VAR par_n: type); { O: description of last parameter


?? PUSH (LISTEXT := ON) ??     ( optional )
<*calls of common decks>
?? POP ??                      ( optional )

<TYPE, CONST and VAR declarations>

<nested procedure/function declarations>
?? SKIP := 4 ?? or ?? EJECT ??
<body of the code>
    .
    .
    .

PROCEND proc_name;
?? OLDTITLE ??
```

For programs, the word PROGRAM is substituted for PROCEDURE and for functions, the words FUNCTION and FUNCEND are substituted for PROCEDURE and PROCEND respectively.

For procedures which have a return value and for functions, the

return value is also described:

```
        .               .
        .               .
        .               .
    VAR par_n: type) { 0: description of last parameter
        return_par: type; { description of return parameter
```

The "?? PUSH (LISTEXT := ON) ?? / ?? POP ??" directives may be used around some or all of the common deck calls.

4.4 COMMON_DECK_LAYOUT

Common decks containing source code contain a one line description of the deck, prologue documentation, the source code, and a *calic to all common decks necessary to compile the source code in isolation (assume a CYBIL module only calls this common deck). Directives to PUSH/POP the LISTEXT toggle must be used to control the listing. LISTEXT is used to ensure that the one line deck description is listed, and may be used to control listing of the called common decks (listing is controlled by the "LO" option on the CYBIL compiler command).

The format for source code common decks is as follows:

```
ggtxxxx
COMMON
?? NEWTITLE := 'ggtxxxx - short description' ??
?? PUSH (LISTEXT := OFF) ??
{ ggtxxxx - short description (single line).
?? POP ??
{
{ Prologue documentation
{ .
{ .
{ .
{ (continuation of prologue)

  < body of common deck >

?? PUSH (LISTEXT := ON) ??  ( optional )
*calic comdeck1             ( common decks necessary to compile )
*calic comdeck2
 ...
*calic comdeckn
?? POP ??                   ( optional )
?? OLDTITLE ??
```

An EJECT may be used with the NEWTITLE directive, for example:

```
?? NEWTITLE := 'ggtxxxx - short description', EJECT ??
```

Note that page width should not be set in common decks.

The format of other common decks is shown with the description of the common deck content in the following sections.

4.4.1 "B" - BINDER DIRECTIVES

The "binder directives" common decks contain Binder directives for the source OPL decks which reference it. This deck is used for the build process and is described in more detail in the Installation Process IDS.

4.4.2 "C" - COMMON DECK CALLS

The "common deck calls" common decks are used to provide a convenient way to call several common decks which are closely related with a single call. These common decks do not call other C type decks, and should not significantly overlap in content.

4.0 CODE LAYOUT
4.4.2 "C" - COMMON DECK CALLS

The format for these decks is as follows:

```
ggCxxxx
COMMON
?? NEWTITLE := 'ggCxxxx - short description', EJECT ??
{ DESCRIPTION:
{   A description of the how the called common decks relate.
{   ... (continue description as necessary).

*calic comdeck1
*calic comdeck2
 .
 .
 .
*calic comdeckn
```

4.4.3 "D" - CONSTANT AND TYPE DECLARATIONS

The "constant and type declarations" common deck contains CYBIL CONST and TYPE declarations, followed by a *calic to all of the declaration common decks necessary to compile this common deck in isolation. VAR declarations are also allowed, but in most cases should appear in a source code module ("M" type decks) rather than in this type of common deck. This type of deck is used by modules dealing with the same types of data. The description in the prologue should explain the relationship between the declarations (ie. what are they for, how are they used, and/or why are they grouped together ?).

"D" type decks follow the format for source code decks described previously.

4.4.4 "G" - GROUP COMMON DECK CALLS

The "group common deck calls" common decks are used to provide a convenient way to call several closely related common decks in another group with a single call. These common decks do not call other G type decks, and should not significantly overlap in content. The G type decks are the same as C type decks, except that they are allowed to call common decks in another group (i.e., an upper layer sharing an interface with a lower layer) without making the interface available to everyone via global (group CM) common decks. The format for these decks is as follows:

4.0 CODE LAYOUT
4.4.4 "G" - GROUP COMMON DECK CALLS

```
ggGxxxx
COMMON
?? NEWTITLE := 'ggGxxxx - short description', EJECT ??
{ DESCRIPTION:
{   A description of the how the called common decks relate.
{   ... (continue description as necessary).

*calic comdeck1  (common decks in another group)
*calic comdeck2
 .
 .
 .
*calic comdeckn
```

4.4.5 "H" - DOCUMENTATION HEADER

A "documentation header" common deck is used for procedures, functions and variables referenced via an XDCL/XREF interface. It contains a prologue block as described earlier (section 2.3.2). This common deck must be called from the module which contains the XDCL defintion and from the XREF common deck. No listing control directives are to appear in this common deck.

```
ggHxxxx
COMMON
{ XDCL/XREF prologue block.
{ ... (continuation of prologue block)
```

4.4.6 "I" - CYBIL INLINE PROCEDURE

The "CYBIL inline procedure" common decks contain procedure declarations which may be expanded inline as part of the calling modules rather than being called through an XDCL/XREF interface. Internal inline procedures may occasionaliy be the most practicai way to implement a "module" (in the Structured Design sense) due to performance and/or scope considerations. A procedure implemented in this fashion must not be dependent on the static chain (ie., it must be completeiy self contained).

"I" type decks follow the format for source code decks described previously.

4.4.7 "L" - LINKER DIRECTIVES

The "linker directives" common decks contain Linker directives for the source OPL decks which reference it. This deck is used for the build process and is described in more detail in the Installation Process IDS.

4.4.8 "T" - TYPE IDENTIFIERS

The "type identifiers" common decks contain type identifiers and calls to B and L type common decks. This deck is used for the build process and is described in more detail in the Installation Process IDS.

4.4.9 "X" - CYBIL XREF DECLARATION

The XREF declaration common deck contains a CYBIL XREF declaration followed by a *calic to all of the decks necessary to compile this declaration in isolation (assume a CYBIL module only calls one XREF declaration common deck). This type of deck is used by modules accessing procedures, functions or variables defined (with the XDCL attribute) in another module.

X type decks follow the format for source code decks described previously.

The prologue documentation is replaced by a call to a documentation header deck, ie.:

```
{
*call ggHxxxx
```

5.0 CODING

When coding, always consider the implications of debugging, modification and maintenance; structure code to make these tasks easier.

5.1 INTERFACES

In general, interfaces between modules should be procedures or functions, not XDCL/XREF variables. The use of explicit parameters is more illuminating than references to global data structures.

The conventions to be used for procedure and function interfaces are defined in the CYBIL Implementation Dependent Handbook.

The DCNS Common Subroutines are to be used instead of self-tailored system interface routines (ie. buffer common routines should be called rather than direct manipulation of buffer descriptor and data fields). Also, when a procedure or function is defined with the XDCL attribute, a common deck containing the XREF declaration for the routine must be defined and all calling routines must reference this common deck. This will minimize the exposure to system specific coding dependencies, eliminate redundant work and increase maintainability.

Declarations which define a specific interface (ie. layer 3A to layer 3B) should be grouped into a common deck, thus providing a clear, concise interface description.

5.2 PROGRAMS/PROCEDURES/FUNCTIONS

5.2.1 PURPOSE

Procedures and functions should be used for two purposes:

- To provide common subroutines within a module.

- To structure the program, thereby making the function of the program obvious at a high level.

5.2.2 LENGTH

In most cases, the length of a routine should be kept down to one or two pages. Anything longer becomes difficult to read, understand and therefore maintain.

5.2.3 COMPLEXITY

The more complex a routine is, the more liable it is to be a source of errors and difficult to implement, modify and maintain. Cyclomatic complexity is a concept which has gained some acceptance within Control Data Corporation. ACSD defines the cyclomatic complexity of a module as the sum of "IF", "WHILE", "FOR" and "REPEAT" statements + 1. For a more detailed analysis of the computation of complexity, see "Structured Testing" by Thomas J. McCabe.

In the interest of limiting the complexity of routines and therefore increasing readability and maintainability, the following general rule should be followed:

A routine should not have a cyclomatic complexity of more than twenty.

5.3 DATA DECLARATIONS

A declaration should always be declared at the inner-most procedure level possible.

Avoid the use of type INTEGER; few variables require subranges that large. Ordinal and subrange types of the appropriate size provide better variable range checking and documentation.

5.4 #LOC FUNCTION

Avoid use of the #LOC function. Code should not be memory location dependent.

5.5 CASE STATEMENTS

Cover all end cases with ELSE being used to cover "unplanned" cases. If the ELSE clause in the CASE statement is missing, the CYBIL compiler will not generate limit checks for CASE entries outside the range of the specified CASE label.

5.6 EXPRESSIONS

In compound arithmetic, conditional or relational expressions, parenthesis should be used to denote precedence. Do not depend on the language operator precedence rules.

Compound boolean expressions should be constructed such that evaluation of the expression terminates as quickly as possible for the typical case.

Use boolean expressions instead of IF statements to conditionally set a value to TRUE or FALSE.

```
equality := (a=b); (instead of)

IF a = b THEN
  equality := TRUE;
ELSE
  equality := FALSE;
IFEND;
```

6.0 CODE READABILITY

6.0 CODE READABILITY

CYBFORM takes care of most of the formatting necessary to maximize code readability. The following sections describe some other techniques for increasing code readability.

6.1 FORMAT OF STATEMENTS

Structured statement pairs (BEGIN/END, FOR/FOREND, WHILE/WHILEND) and IF/IFEND pairs are hard to match when separated by more than 10 lines of code, or when they are nested.

For structured statements, labels are allowed and should be used in these cases to assist in matching the pairs (as well as for documentation purposes).

For example:

```
/search_symbol_table/
  FOR i := 1 to 10 do
    ...
  FOREND /search_symbol_table/;
```

For IF/IFEND pairs, comments imitating labels may be used. In this case, there should be no blank line following the initial comment (normally, a blank line should be on either side of a stand alone comment).

For example:

```
{ Check command file status.
      IF command_file_status = success THEN
        .
        .
        .
      IFEND; { Check command file status.
```

Compound conditionals in an IF, FOR or WHILE statement must be separated at the OR/AND if the entire statement does not fit on a single line. At the programmer's option, the statement may also be separated at each OR/AND regardless of line length to make the code more readable.

6.0 CODE READABILITY
6.1 FORMAT OF STATEMENTS

For example:

```
IF ((cond_a) AND (cond_b)) {
OR (cond_c) THEN
  .
  .
  .
IFEND;
```

(or)

```
IF ((cond_a) {
AND (cond_b)) {
OR (cond_c) THEN
  .
  .
  .
IFEND;
```

6.2 DECLARATIONS

Where a declaration contains a list of items it may be helpful to list each element on a separate line. Defining ordinals, sets or presetting elements of an array are examples. Comments can be used to force splitting the declaration (when the code is run through CYBFORM).

With descriptive names, a comment with text may not be necessary and blank comments may be used.

For Example:

```
TYPE
  clt$character_class = ( {
    clc$space_character, {
    clc$comment_delimiter_character, {
    clc$string_delimiter_character, {
    clc$digit_character, {
    clc$alpha_character, {
    clc$token_character, {
    clc$digraph_token_character, {
    clc$other_character  {
    );
```

6.0 CODE READABILITY
6.2 DECLARATIONS

Where readability would be greatly enhanced by alignment of code and/or comments, the CYBFORM FMT pragmat may be used to turn off formatting for a specific block of code. The code must still start in the appropriate column, and formatting must be turned back on at the end of the block of code.

```
?? FMT (FORMAT := OFF) ??
{ Mainframe Channel Interface Intertask Message Workcodes.

  CONST
    mci_startup            = 0501(16), { Specific MCI card
    mci_output_complete    = 0502(16), { PP has successful read
    mci_input_received     = 0503(16), { PP has successful write
    mci_data_available     = 0504(16), { Data is available for transfer
    mci_error_encountered  = 0505(16), { An error was found on a write
    mci_shutdown           = 0506(16), { End processing
    mci_statistics         = 0507(16), { Sender requests statistics
    mci_report_statistics  = 0508(16), { Announce statistics response
    mci_regulation_change  = 0509(16), { New regulation level in effect
    mci_timer_expiration   = 050a(16), { Response timer has expired
    mci_log_message        = 050b(16); { Message is to be logged
?? FMT (FORMAT := ON) ??
```

6.3 BLANK LINES

A blank line should precede and follow all stand alone comments. Additional blank lines should be used at the coder's discretion to enhance code readability. Blank lines should be used consistently within a module.

It is helpful to insert blank lines after each RETURN, CYCLE or EXIT statement to indicate a break in the program flow.

A1.0 CODING EXAMPLES

A1.0 CODING_EXAMPLES

A1.1 PROCEDURES

```
?? NEWTITLE := 'Bypass Configuration Command Processors' ??
?? NEWTITLE := 'PROCEDURE [XDCL, #GATE] cmd_bypass_configuration', EJECT ??

{ PROCEDURE cmd_bypass_configuration
{
{ PURPOSE:
{   Process the bypass_configuration command.
{
{ DESCRIPTION:
{   The global variable bypass_config_flag is set to indicate to
{   the Configuration Procurer that an operator wishes to enter
{   the configuration manually.
{
{ COMMAND FORMAT:
{   BYPASS_CONFIGURATION (BYPC)
{
{ GLOBAL DATA MODIFIED:
{   bypass_config_flag - set to indicate that configuration file
{         processing should be bypassed.
{   command_source - the address of the user that issued the
{         command is saved
```

see p. A

```
PROCEDURE [XDCL, #GATE] cmd_bypass_configuration ( {
      parameter_list: ost$string;
    VAR pvt: clt$parameter_value_table;
    VAR response_code: condition_code;
    VAR response: clt$status);


?? PUSH (LISTEXT := ON) ??
*calic metmdu
*calic mexgsa
?? POP ??


  VAR
    error_msg: [STATIC] string (39) := {
      'No parameters expected after command.' CAT
      mec$end_of_line,
    task: task_ptr;
?? EJECT ??
```

A1.0 CODING EXAMPLES
A1.1 PROCEDURES

```
{ Begin CMD_BYPASS_CONFIGURATION.

    response.condition := NIL;

{ Check for parameters.  No parameters are allowed.

    IF parameter_list.size <> 0 THEN
      response_code := par_err_ccode;
      response.normal := FALSE;
      gen_data_field (response.condition, ^error_msg, #SIZE
            (error_msg), char_octet);
      RETURN;

    IFEND;

    task := NIL; { indicate current task_ptr to be used
    get_source_address (command_source, task);
    bypass_config_flag := TRUE;

    response_code := ok_ccode;
    response.normal := TRUE;

  PROCEND cmd_bypass_configuration;
?? OLDTITLE ??
?? NEWTITLE := 'PROCEDURE [XDCL, #GATE] cmd_bypc', EJECT ??

{ PROCEDURE cmd_bypc
{
{ PURPOSE:
{   This is the command processor for the bypass_configuration
{   command alias, bypc.
{
{ DESCRIPTION:
{   Cmd_bypass_configuration is called to process the command.
{
{ COMMAND FORMAT:
{   Refer to the bypass_configuration command description.


  PROCEDURE [XDCL, #GATE] cmd_bypc ( {
        parameter_list: ost$string;
    VAR pvt: clt$parameter_value_table;
    VAR response_code: condition_code;
    VAR response: clt$status);
?? SKIP := 4 ??
{ Begin CMD_BYPC.

    cmd_bypass_configuration (parameter_list, pvt,
          response_code, response);
```

A1.0 CODING EXAMPLES
A1.1 PROCEDURES

```
  PROCEND cmd_bypc;

?? OLDTITLE ??
?? OLDTITLE ??
```

A1.0 CODING EXAMPLES
A1.2 "C" TYPE COMMON DECK

A1.2 "C" TYPE COMMON DECK

```
CMCTREE
COMMON
?? NEWTITLE := 'CMCTREE - Tree Management Definitions', EJECT ??
{ DESCRIPTION:
{   CMCTREE contains calls to the common decks which contain
{   tree management definitions.

*calic CMDTTRE
*calic CMXPEXC
*calic CMXPFFN
*calic CMXPFIN
*calic CMXPFNC
*calic CMXPFNF
*calic CMXPFNX
*calic CMXPGRD
*calic CMIPINT
*calic CMXPPIC
?? OLDTITLE ??
```

A1.3 "D" TYPE COMMON DECK

```
CLDINT
COMMON
?? PUSH (LISTEXT := OFF) ??
{ CLDINT - Convert String to Integer Result.
?? POP ??
{
{ DESCRIPTION:
{   The TYPE declaration for the result of the CLP_CONVERT_
{   INTEGER_TO_STRING procedure is defined.

  TYPE
    cltsinteger = record
      value: integer,
      radix: 2 .. 16,
      radix_specified: boolean,
    recend;
```

A1.0 CODING EXAMPLES
A1.4 "H" TYPE COMMON DECK

A1.4 "H" TYPE COMMON DECK

```
CLHCS2I
COMMON
{ PROCEDURE clp_convert_string_to_integer ALIAS clpcs2i
{
{ PURPOSE:
{   This procedure converts the string representation of an integer
{   to an integer.  The string representation may contain a leading
{   sign (+ or -) and/or a trailing radix enclosed in parentheses.
{
{ CALLING FORMAT:
{   (*calic clxcs2i)
{   CLP_CONVERT_STRING_TO_INTEGER (STR, INT, STATUS) or
{   CLPCS2I (STR, INT, STATUS)
{
{   STR: (input)  This parameter specifies the string to be converted.
{   INT: (output)  This parameter specifies the converted integer
{         value along with the radix in which the integer was
{         represented.
{   STATUS: (output)  This parameter specifies the status of the
{         request.
```

A1.5 "X" TYPE COMMON DECK

```
CLXCS2I
COMMON
?? NEWTITLE := 'CLXCS2I - Convert String to Integer' ??
?? PUSH (LISTEXT := OFF) ??
{ CLXCS2I - Convert String to Integer.
?? POP ??
{
*call clhcs2i

  PROCEDURE [XREF] clp_convert_string_to_integer ALIAS 'clpcs2i' ( {
          str: string (*); { I: string to be converted
      VAR int: clt$integer; { O: converted integer value and radix
      VAR status: clt$status); { O: request status

?? PUSH (LISTEXT := ON) ??
*calic cldint
*calic cldstat
?? POP ??
?? OLDTITLE ??
```

A1.6 "X" TYPE COMMON DECK

```
CLXCS2I
COMMON
?? NEWTITLE := 'CLXCS2I - Convert String to Integer' ??
{ CLXCS2I - Convert String to Integer.
?? PUSH (LISTEXT := ON) ??
{
*call clhcs2i

  PROCEDURE [XREF] clp_convert_string_to_integer ALIAS 'clpcs2i' ( {
          str: string (*); { I: string to be converted
      VAR int: clt$integer; { O: converted integer value and radix
      VAR status: clt$status); { O: request status

?? PUSH (LIST := OFF) ??
*call cldint
*call cldstat
?? POP ??
?? POP ??
?? OLDTITLE ??
```

B1.0 ABBREVIATIONS AND ACRONYMS

Standard industry abbreviations and programming language names may be used even though they are not included in the following appendix.

A-A	Application to Application
AP	Application Process
BERP	Background Processing
BERR	Bus Error
BTF	Batch Transfer Facility
BVT	Batch Virtual Terminal
CD	Collision Detection
CDNA	Control Data Network Architecture
CDCNET	Control Data Distributed Communications Network
CEP	Connection Endpoint
CEPID	Connection Endpoint Identifier
CIB	Communications Interface Module Instruction Block
CIM	Communications Interface Module
CMM	Cache Memory Module
CPDU	Channel Protocol Data Unit
DCB	Device Control Block
DCI	Data Concentrator Interface
DCNS	Distributed Communications Network Software
DI	Device Interface
DVM	Device Manager
ESCI	Ethernet Serial Channel Interface
FRU	Field Replaceable Unit
FTP	File Transfer Protocol
HDB	HDLC Downline Interface Block
HDLC	High Level Data Link Control
HSB	High Speed Bus
HTB	HDLC Trunk Identification Block
HUB	HDLC Upline Interface Block
ICB	Internal Control Bus
IDU	Interface Data Unit
IOSS	Input/Output Subsystem
ISB	Internal System Bus
ITB	Internal Transfer Bus
ITF	Interactive Transfer Facility
IVT	Interactive Virtual Terminal
LIM	Line Interface Module
MCI	Mainframe Channel Interface
MDI	Mainframe Device Interface
M-E	Management Entity
MPB	Master Processor Board
MTI	Mainframe Terminal Interface
NDI	Network Device Interface
NHP	Network Host Products
PDU	Protocol Data Unit

PMM	Private Memory Module
SAP	Service Access Point
SAPID	Service Access Point Identifier
SDU	Service Data Unit
SMM	System Main Memory Module
SSR	Stream Service Routine
SVM	Service Module
TDI	Terminal Device Interface
TWA	Two Way Alternative
TWS	Two Way Simultaneous
URD	Unit Record Device
URI	Unit Record Interface
VTP	Virtual Terminal Protocol

C1.0 ERROR MESSAGE GUIDELINES

Error messages represent a very important, though often neglected, interface between software and the user. Proper attention to producing polite, correct and clear error messages can do a lot toward improving the overall usability of the system. The following general principles are to be observed in designing error messages.

- Messages must be formattable for 72 character displays.

- The purpose of an error message is to inform the user how to correct a problem.

It helps to view error messages as prompts: not "this is what you did wrong" but "this is how to do it right." Messages should be phrased positively. The words "ILLEGAL", "INVALID", and "SYNTAX" are specifically not permitted in messages. Of course, there are other ways to phrase unhelpful, negative messages; but these three words are singled out for extinction for being so frequently seen in the company of usability offenders.

- A single message should diagnose a single error.

For example, if the meaning of message is "more than seven characters or leading non-alphabetic character or null identifier" it should be three messages. Usually, the code must make three separate tests, so it is easy to be precise. An exception is when a common deck returns an error status which could have resulted from several different conditions.

- An error message is friendly if it is business-like and informative.

Cute, funny, or flippant messages are to be avoided, as they seldom diagnose accurately and always wear quickly. Messages should be directed at the process and not the person.

- Messages must be written plainly, using terms already known to the user.

Messages should use terms which are either self-defining or natural to the process. All words should be part of the external user interface, like "file name" instead of "LFN" (unless LFN is an external parameter).

- Messages must be written in English.

Messages should follow normal rules for English grammar and punctuation, although "pidgin English" (the omission of selected subjects, verbs or objects in the interest of brevity where the

meaning is clear) is acceptable. Use upper and lower case as they would normally be used in the English language. Upper case should be used to distinguish variable parts from normal English words. Ending punctuation should be used. It indicates to the user that the message is not continued on the next line and adds to the readability of the message. Messages should not be written in octal, or in other forms of scientific notation. Note that the asterisk is not an English punctuator.

- Messages should be self-contained.

If you need to tell a story, tell the whole story. Avoid references, as they are difficult to keep up-to-date and are often no more helpful then a good one-line message would be.

- Error messages should point directly to the source of the trouble.

For example, "File not found." is better put as "File XYZ not found.", "Expecting comma or period after ABC." is much clearer than "Syntax error.". In general, the technique of echoing back part of the user input as part of the message is better than the use of internal names or parameter keywords which the user may not recognize.

- Interactive error messages should appear as soon as possible after an error is committed.

Each interactive input should be completely and fully validated as soon as it is received. In no event should a user be led down the garden path to enter a long series of input only to be advised that it is all wrong because the first part was wrong.

- No messages at all should appear for trivial, correctable errors, nor should they be errors.

Errors such as missing or redundant terminators should not be errors at all. If a reasonable assumption can be made as to the intent of an input, it should be acted upon as though it were "valid". No error diagnostic should be produced for these cases. If it is not perfectly clear what assumption was made, the assumption was probably not reasonable to begin with.

- An error message must clearly signal that an error has occurred.

An error message must not be phrased in such a way as to be confused with a merely informative message. Also, a message should indicate the gravity and extent of the error, as when an error in a list inhibits processing of the remainder of the list.

PROCEDURE_INTERFACE_CONVENTIONS

INTRODUCTION

The purpose of this section is to describe the conventions that should generally be used by designers of procedural interfaces.

PURPOSE

The purpose of the following conventions is to achieve a software system which exhibits the beneficial characteristics of being understandable, reliable, efficient, maintainable, etc.

GENERAL_PHILOSPHY

- Select simple straightforward interfaces. Complex interfaces, those whose description contain 'and', 'or', and conditional clauses, impair understanding of the function. If there is not an evident choice between a single complex interface and multiple simple interfaces, choose the simple interfaces.
  - A single interface encompassing multiple intrinsic functions, which cannot be performed in conjunction with one another, unduly increases validation overhead. A simple interface for each intrinsic function is preferred.
  - If the intrinsic functions encompassed by a single interface require different degrees of user privilege, each intrinsic function should be a single simple interface.
  - The combination of multiple intrinsic functions into a single interface is practical when the functions can logically be performed in conjuction with one another.
- Input parameters should be validated early in the processing when the correlation between the potential error and the actual parameter is readily identifiable. This aids in ensuring that diagnostics accurately reflect the cause of the error.
- Wherever feasible, delegate the error prognosis to the requestor (i.e., return control to the requestor with accurate information when an error is detected).
- Refrain from exposing internal structures or concepts via externalized interfaces. Before externalizing internal structures or concepts rate the probability of change and the user consequences (re-code, re-compilation, etc.) if in fact the externalization changes.

INPUT PARAMETER CONVENTIONS

Input parameters in the following conventions are formal parameters in the Xref procedure declaration.

o Declare all input parameters to be <value params>.

  - If for any reason input parameters are declared as <reference params>, the actual parameters must be moved to local automatic variables prior to validity check and subsequent usage. Further, all input parameters declared as <reference params> must be moved before any validation or usage occurs.

o All input parameters must be checked for validity with explicit language statements prior to use. In fact all input parameters should be validated before any parameter is used.

o Input parameters which specify subfunction or function option should be discrete parameters (i.e., should not be a field of a record).

PARAMETER TYPING - CYBIL USAGE

Parameter types are declared in terms of the CYBIL pre-defined types or type identifiers which resolve to the pre-defined types.

o The first inclination should be to declare parameter types as type identifiers, declaring their ultimate types with type declarations.

  - The language and general ease of use dictates that ordinal, array, and record parameter types be declared as type identifiers.

  - For parameter types other than pointer and cell, before selecting the pre-defined types consider the following: 1) if the concept of the parameter is used by more than one external interface, use a type identifier; 2) if the parameter type has any significant probability of change, use a type identifier; and 3) if the parameter identifier cannot accurately convey the purpose and intent, use a supportive type identifier.

o Ordinal or boolean parameter types are preferred over integer or integer subrange when declaring subfunction or option parameters. If the scope of a boolean parameter type has any significant probability of exceeding binary, that parameter type should be declared as an ordinal type.

o Take advantage of the self documenting aspect of ordinals by using descriptive ordinal type identifiers and ordinal constant identifiers on parameters.

o An ordinal type should be consistent within itself, that is, there should

be an evident relationship among the ordinal type identifier and ti ordinal constant identifiers.

- o An ordinal type should support only one concept.

- o Before utilizing an ordinal subrange in an interface, consider defining a new ordinal type. If an ordinal subrange is the appropriate choice, declare that subrange as a type identifier.

- o Integer subrange is preferred over integer when declaring numeric parameters. Further, the integer subrange should be declared as a type identifier and the bounds of the subrange should be specified with descriptive constant (CONST) declarations. The low bounds, if zero or one, need not be specified with constant declarations.

- o Use a constant (CONST) declaration to specify length of string type parameters.

- o Set type provides a mechanism by which multiple subfunctions or options may be discretely specified with a single parameter. This use of set type is preferred over the use of codes each of which specifies a combination of subfunctions or options.

- o Array type parameters will provide a convenient, useful, efficient interface in the bounds of the convention objectives if the following criteria is achieved

  - The function can be logically performed on multiple arguments of the same type (array components) with one request; or the function can logically generate multiple values of the same type with one request.

  - Each array component can be acted upon (or generated) in absence of all other components.

    - The result of the function relative to one component has no effect on the result of the function for any other component.

    - The order of the components has no bearing on the individual results.

- o Record type parameters provide a convenient, useful interface in the bounds of the convention objectives if the record can be thought of (in the user's sense) as a single unified entity (i.e., no field of the record has particular significance in absence of any other field). If a field does not meet this criteria, it should be a discrete parameter.

  - A record parameter type will simplify interfaces and be convenient when the record is also a parameter of other external interface procedures and does not require user intialization or manipulation of contents - the user need only be concerned with the concept of the parameter, its structure and contents are transparent.

- Each field should have an evident consistent relationship with the other fields of the record. Merely being parameters of a function does not establish the unified relationship.

- If a field by itself has particular significance, that field should be a discrete parameter. Fields which are subfunction or option parameters to a function have such significance and should be discrete parameters.

- A record type parameter should not contain fields which are superfluous to the execution of a function. Each field of an input parameter record should be essential to the execution of the function (i.e., each field should be a required argument). Each field of an output parameter record should contain a value returned by the function.

  - Record type parameters may contain superfluous fields if the fields are present for symmetry with other functions supporting the same concept. Use of this direction to justify superfluous fields should be minimized - superfluous fields will impair user understanding and result in excessive re-work at maintenance and extension time.

  - System architecture may dictate that some seemingly superfluous fields appear in a record to reserve space for data used internally by a function in support of other functions relating to the same concept - this is justifiable.

- A record type parameter should be solely an input parameter or solely an output parameter (i.e., a record should not contain some fields which are input parameters and other fields which are output parameters).

o Input parameters should not be pointers (CYBIL pointer type) to internal objects - validation of the pointer object would be virtually impossible.

o Pointers to internal objects (output parameters of CYBIL pointer type) must not be returned to the user - unnecessary exposure of internal data will result if such pointers are returned.

o Pointer type formal parameters should be declared only when the pointer object of the actual parameter can take one of several types (i.e., the pointer object type is not known at compile-time, but is resolved at execution-time). The formal parameter pointer type should ultimately resolve to '^cell'.

o Packed structures, adaptable types, and bound variant records have some applicability in external interfaces, but their use should be the exception rather than the norm.

PROGRAM LIBRARY CONVENTIONS

DECK NAMING CONVENTIONS

Deck names have the following format:

PPCZZZZ

where

PP = two character product identifier
C = one character indicating deck class
ZZZZ = one to four character mnemonic for uniqueness within product

Allowable codes for deck type are as follows:

M = CYBIL code module
X = CYBIL xref declaration (common deck)
D = CYBIL type and const declarations (common deck)
H = Documentation header (common deck)
I = CYBIL internal in-line procedure (common deck)

Note that decks of type M must consist of exactly one module (compilation unit).

When converting to the source code utility (SCU) all XREF declarations, documentation headers and module decks can be renamed. The new deck name will have the same three character prefix but the suffix (ZZZZ) can be the full name (up to 28 characters) of the item contained in the deck.

COMMON DECK USAGE

Common decks are restricted to four classes of usage:

- XREF declarations to be used by modules accessing procedures or variables defined in another module.
- TYPE and CONST declarations to be shared by modules dealing with the same data types or constants.
- Documentation header text describing an interface. A common deck of this type must be called from the module which contains the XDCL definition of the interface being described.
- Procedure declarations which may be expanded in-line as part of calling modules; as opposed to being called through an XDCL/XREF interface. Internal in-line procedures may occasionally be the most practical way to implement a "module" (in the Structured Design

sense) due to performance and/or scope considerations. All comm decks of this type are considered internal interfaces and must be documented accordingly. A procedure implemented in this fashion must not be dependent on the static chain, i.e. it must be completely self-contained.

COMMON_DECK_CONTENT

DOCUMENTATION HEADER

Procedures

The procedure documentation header consists of CYBIL comments which describe the procedure, its calling sequence and parameters. The general format for the procedure documentation header is as follows:

```
   123456789012345...
 1)(}
 2)(   The purpose of this request is to ...
 3)( whatever this request does.
 4)(}
 5)(      XXP$REQUEST_NAME (FIRST_PARAM, ...,
 6)(        LAST_PARAM)
 7)(}
 8)( FIRST_PARAM: (input) This parameter specifies ...
 9)(       whatever this parameter specifies.
10)(}
11)( LAST_PARAM: (output) This parameter specifies ...
12)(       whatever this parameter specifies.
13)(}
```

where:

line 1: blank comment line
line 2: indent 4: describe the purpose of the request
line 3: indent 2: for purpose continuation, if necessary
line 4: blank comment line
line 5: indent 8: request calling sequence; use all capital letters; parameter names must be the same and must be in the same order as in the XREFed procedure declaration
line 6: indent 10 for parameter continuation if necessary
line 7: blank comment line
line 8: indent 1: describe first parameter; specify whether it is input, input-output, or output
line 9: indent 8: for parameter description continuation, if necessary
line 10: blank comment line separates each parameter
line 13: blank comment line

Also, when listing parameters one should strive to list all input parameters first followed by input-output parameters followed by all output parameters unless there is an obvious symmetry with other requests that

would be violated. The status parameter, if present should always be the last parameter on every request.

Data Structures

Each data structure will include a documentation header consisting of CYBIL comments which describe what the structure is for and how it is used. The general format is as described for the "Purpose" section of the procedure header.

XREF DECLARATION COMMON DECK

The XREF declaration common deck contains a CYBIL XREF declaration followed by a *calic to all of the TYPE or CONST declaration common decks ("D" decks) necessary to compile this declaration in isolation (assume a CYBIL module only calls one XREF declaration common deck).

It is very important that all XREF declaration common decks perform *calic's (instead of *call) to necessary decks. This prevents duplicate definitions of identifiers in the caller's CYBIL module.

Example:

```
AMXREWD
COMMON

     PROCEDURE [XREF] ampsrewind(file_identifier:
               amt$file_identifier;
               wait:ost$wait;
          VAR status:ost$status);

??  PUSH (LIST := OFF, LISTEXT:=ON ) ??
*calic amdfid
*calic osdwnw
*calic osdstat
??  POP ??
```

TYPE / CONST DECLARATION COMMON DECK

The TYPE / CONST declaration common deck contains CYBIL TYPE and/or CONST declarations followed by a *calic to all of the declaration common decks necessary to compile this common deck in isolation.

It is very important that the declaration common decks perform *calic's (instead of *call) to common decks. This prevents duplicate definitions of identifiers in the caller's CYBIL module.

Example:

```
AMDNAME
COMMON

TYPE
  amt$local_file_name = ost$name;

*calic osdname
```

EXAMPLE DECK

In order to be certain that interfaces provided for the end-user or other functional areas are specified accurately and consistently, each contributor should produce an example compilation unit that includes references to all type and procedure declarations he/she is responsible for and an example of the usage of each interface. By compiling all declarations, the checking logic of the compilers will aid accuracy and consistency; by trying examples of the interface, the contributor will gain a feeling for the efficacy of the interface.

CYBIL CODING CONVENTIONS

This document specifies the CYBIL coding conventions suggested for the CYBIL users. There are several general aims of coding conventions which underlie all of the specific proposals that follow:

1. There are a variety of routine, mundane aspects associated with writing programs: a set of coding conventions remove from the programmer trivial decisions relating to module format, name generation, etc. thereby leaving more time to concentrate on important matters.

2. The primary purpose of documentation and the readability of source code is to help someone other than the developer understand what is going on.

3. During the lifetime of a large software product like an operating system or a compiler, the average developer will come in contact with a large number of modules written by and maintained by many other programmers. A consistent set of coding conventions helps the programmer "feel at home" with a new module and therefore is able to begin doing useful work sooner.

4. To as great an extent as reasonable, all coding conventions should be generated and reinforced by automated methods.

5. Source code is the ultimate documentation of any program, particularly a program written in a higher level language such as CYBIL. Therefore, in all CYBIL programming, a consistent emphasis should be placed on producing lucid, readable, self- documenting code.

6. All commentary in the source code should be written so that it: a) only provides information not readily apparent from reading the code and b) is of a sufficiently algorithmic nature such that it rarely, if ever, becomes obsolete as changes are made to the code.

USAGE OF A SOURCE CODE FORMATTER

The major software tool for generating and enforcing CYBIL coding conventions should be the source code formatter (CYBFORM).

USE OF CYBIL

- Use block structure to articulate program structure: a declaration should always be declared at the "lowest" level possible.

- Do not use the static chain: in general a procedure should only reference arguments, its own automatic variables and static variables.

- In general, interfaces between modules should be procedures or functions, not XDCL/XREF variables.
- Always use label names that describe the process being performed by the structured statement to which the label refers.
- Always repeat the label in the terminating statement of a structured statement (the formatter will do this): e.g.:

```
/search_symbol_table/
  for i := 1 to 10 do
    ...
  forend /search_symbol_table/;
```

- In general avoid the use of type INTEGER; few variables require subranges that large.
- In declarations of procedure parameter lists, always separate each formal parameter with a semicolon marking each with a VAR or "absence of VAR" as appropriate.
- Always declare all input parameters before all output parameters unless there is an obvious symmetry that would be disturbed.
- Cover all end cases. CASE statements should cover all statements with ELSE being used to cover "unplanned" cases.
- Procedures and functions should be used for two purposes: 1) "subroutines", 2) to "structure" the program thereby making the function of the program obvious at a high level.
- Arguments to procedures should also be used for two purposes: 1) "subroutine parameters", 2) as documentation which allows the reader to see all data referenced by the procedure by looking at the procedure call statement. In the latter case, the formal and actual parameter names should be the same.
- Trailing comment delimiter of '}' should be used whenever reasonable: i.e., use of EOL as a comment delimiter is discouraged.
- In compound arithmetic, conditional or relational expressions, use parenthesis to denote precedence. Do not depend on the language operator precedence rules.
- Avoid the #LOC function like the plague.

USE OF THE ENGLISH LANGUAGE

The key to making programs readable is the usage of meaningful, non-cryptic English names for all CYBIL constructs; specifically:

- When naming type identifiers and record fields, particularly field consider the way the name will look in the code, not the declaration; e.g.:

```
TYPE
  program_descriptor = record
    load_map: load_map_options,
  recend,
  load_map_options = record
    file_name : file_name,
    options : (all,nothing),
  recend;
VAR
  my_program : program_descriptor;
        ...
  my_program.load_map.file_name := "LOADMAP";
```

- Procedure and function names should describe the process the procedure performs.

- Labels should always describe the function being performed by the structured statement to which they refer; e.g.:

```
/search_symbol_table/ {instead of}
/l1/
```

- Labels are a powerful documentary aide and their usage is encouraged.

- Booleans should always describe the TRUE condition; e.g.:

```
if file_is_open then {instead of}
if file_switch then
```

CYBIL_NAMING_CONVENTION

It cannot be emphasized too strongly that names should be chosen for how they will read in the code body of a procedure, not how they look in the data declaration. This is particularly true of variables and field names in type declarations.

The system naming convention for the user interfaces is described in the System Interface Standard (SIS). That is also the convention for linkage (entry-point or external) names. However, local names should use no convention other than English. For convenience, selected portions of the SIS naming conventions are reproduced below:

System global names will be generated according to the following convention:

PPCSXXX...

where:
- PP = is a two character product identifier for the owner of this name.
- C = identifies the class of the name.
- $ = is the special character '$'.
- XXX = a meaningful English expression or abbreviation that describes or denotes the purpose of the item being named.

Class of Names:

- C - constant
- E - exception condition name
- F - file
- M - module
- P - procedure
- S - section
- T - type
- V - variable

MODULE AND PROCEDURE DOCUMENTATION

Standard documentation for each module and each XDCLed procedure or function within a module should be provided. The procedure documentation is also encouraged for local procedures and functions as well. Care should be taken to minimize commentary becoming outdated as changes are made to the code.

```
MODULE <module identifier>;

{  PURPOSE:
{    This should contain the purpose of the module and the
{    reasons for grouping these declarations in the module rather
{    than the purpose of each procedure.
{  DESIGN:
{    This should contain an overview of the module design; i.e.,
{    an outline of how it works in general terms.  Usage of
{    specific variables or procedure names is discouraged in this
{    description.

<procedure or function declaration>;

{  PURPOSE:
{    This should describe the process the procedure or
{    function performs rather than the method used.
{  NOTE:
{    This should contain information of interest to the
{    user or maintainer.
```

TITLE PRAGMAS

Each module should be titled in the following way:

```
<major product identifier>[:<component identifier>...]
<sp><sp>[[XDCL]]<procedure identifier>:<section identifier>
```

for example:

```
NOS : task establisher
  [XDCL] pmp$establish_task
```

COMMENTING CONVENTIONS AND GUIDELINES

In general, comments should be standalone blocks describing why or what a series of CYBIL statements are doing. Care should be taken not to use comments that will become outdated by detailed changes to the code. The basic concept behind comments should be to provide nonredundant information. Comments should be preceded and followed by a blank line and start in the first available source character on the line. Again, remember that the purpose of comments is to help someone other than the original developer of the module understand what the module is doing.

PROCEDURE AND DATA ATTRIBUTE COMMENT CONVENTIONS

Comments should also be used to convey software or system attributes which are not discernable from CYBIL declarations. These comments should be concise and abut CYBIL declaration constructs rather than being standalone blocks.

EFFICIENCIES

This section lists a group of programming tips to help the user make better utilization of the CYBIL development environment. As such, it is not an exhaustive list and will be added to as additional hints become known. The CYBIL Project would appreciate any other information which may assist the usage of CYBIL.

These ideas are guidelines, they should be followed only when clarity of code is not compromised.

SOURCE_LEVEL_EFFICIENCIES

GENERAL

- There is a significant amount of overhead associated with any procedure call. If a procedure is being called in a looping construct, it may pay to call the procedure once and put the loop tests inside the called procedure.
- References to variables via the static chain in nested procedures cause an overhead associated with that reference. In general, a procedure should only reference static variables, arguments and its own automatic variables.
- A copy is currently being made of all value parameters. This implementation is subject to change.
- Assignment of records is done with one large move, while record comparison is done field by field. Therefore, all other things being equal, it is best for performance reasons, to organize fields within records with the most likely non-conforming fields first.
- Move structures rather than lots of elementary items. This may require structuring the elements together especially for this purpose.
- Reference to adaptable structures are slower than references to fixed structures because the adaptable has a descriptor field which must be accessed.
- References to fields within a record require no execution penalty.
- Repeated references to complex data structured (via pointers or indexing operations) can be made more efficient by pointing a local pointer at the structure and use it to replace the complex references.
- Inappropriate use of the null string facility can be an expensive NOOP.

- Initialization of static variables incurs no run time overhead.
- If a record is being initialized with constants at run time it is often more efficient to define a statically initialized variable of the same type and do record assignment.
- A packed structure will generally require less space at the possible cost of greater overhead associated with access to its components. This is because elements of packed structures are not guaranteed to lie on addressable memory units.
- When organizing data within a packed structure it is more space efficient to group bit aligned elements together.
- The STRING data type is a more efficient declaration than a PACKED ARRAY OF CHAR.
- When considering alternative data structures for homogenous data the user should first consider ARRAYs, then SEQuences and finally HEAPs.
- When considering alternatives between the HEAP and SEQuence storage types, the following should be considered. The HEAP is the more inefficient mechanism requiring the greatest overhead in terms of space requirements and the more execution overhead. SEQuences are the more efficient in terms of both storage and execution overhead.
- The NEXT and RESET statements as used on sequences and user heaps are implemented as inline code. Whereas the implementation for ALLOCATE and FREE is a procedure call to run time library routines.
- Space in a heap is consumed only when an ALLOCATE statement is executed. In addition to the space ALLOCATEed by the CYBIL program, a header is added to maintain certain chaining information. For this reason, ALLOCATEing small types incurs a large percentage overhead.
- Code for the PUSH statement is generated inline and, as such, is considerably faster than an ALLOCATE and FREE combination.
- For efficiency and maintainability reasons the use of #LOC should be avoided.
- When a definition contains a number of 'flags' or attributes, the following should be considered when chosing between BOOLEANs or a SET type:
  - If the record is not packed the SET will reduce the size of the definition
  - Any sub-set of the attributes of a SET can be tested at once.
  - If a single element test is desired an unpacked BOOLEAN is slightly more efficient than a SET.

- Usage of boolean expressions is more efficient than IF statements. For example, use:

```
equality := (a=b);
```

  Do not use:

```
IF a=b THEN
  aquality := TRUE;
ELSE
  aquality := FALSE;
IFEND;
```

- Rather than coding long IF sequences a CASE statement should be considered when using a proper selector.

- Compound boolean expressions should be ordered such that the first condition is the one which has the highest probability of terminating the condition evaluation for the nominal case.

- Compile time evaluation of expressions involving constants produces better object code if all constants (at the same level) in the expression are grouped together. For example, the expression:

```
X := 5 * Y * C * 2 ;
```

  will produce object code using two constants (5 and 2) and two variables (Y and C). If the expression is rewritten:

```
X := 5 * 2 * Y * C;
```

  with the constants together, the compiler (at compile time) will combine the expression "5 * 2" into the constant "10" and produce object code to evaluate the expression using only one constant (the ten) and two variables (Y and C).

- When doing divide by a power of two on a positive integer subrange a shift instruction can be generated. Because a shift instruction instruction tends to be considerably faster than a divide instruction it is a benefit to define positive integer subranges.

- Range checking code requires additional storage space and is time consuming. One can eliminate all generated range checking code by setting "CHK=0" on the call statement (or ??SET(CHKRNG:=OFF)?? in the source program). Setting CHK=0 on the call statement, while debugging programs, is not recommended since legitimate program errors may not be diagnosed. A better approach is to request range checking on the call statement (or in the source program) and then minimize, using good programming practice, the amount of checking code generated. Consider the following program segment:

```
TYPE
  a = 0..10;
VAR
  index,y: a,
  x: array [a] of integer;
  .
y:=5;.
index:=y:
x[index] :=3;
```

Since variables "index" and "y" are defined to be of type "a" (the subrange 0..10) the assignment "index :=y;" will not (and need not) be checked for proper range even if range checking is requested. Similarly, the statement "x[index] :=3;" will not (and need not) contain range checking code. If variables "y" and "index" were declared to be INTEGER (or some type other than the subrange 0..10) range checking code would be required.

- Any timed executions should be run after the CYBIL code has been built with checking code turned off.

- Certain conversion functions (i.e.,$INTEGER,$CHAR,etc.) require no execution time overhead.

- The code generated for STRINGREP is a call to a run time library routine.

- A file should not be opened before it is needed. As soon as a file is no longer needed, it should be closed. An overhead is involved in opening & closing files. Therefore, unnecessary opens & closes should be avoided.

CC EFFICIENCIES

- Pointers to strings are inefficient because the string may, in general, begin at any character boundary. These pointers may be created explicitly by assignment statements or implicitly by supplying a string as an actual parameter for a call by reference formal parameter. If possible, align strings so that they begin on a word boundary.

- Run time routines are called for the string operations of assignment & comparison when:

  1) Neither string is aligned or,
  2) Lengths are known and unequal or,
  3) Either or both lengths are unknown at compile time.

  Otherwise the faster inline code is generated.

- It is possible to modify the buffer size used by the CYBIL I/O package.

For an explanation see the ERS for CYBIL I/O (ARH2739). If there are very few accesses to a file, it may be best to select a small buffer, since overall field length will be reduced, thereby increasing total system throughput by decreasing swap rates, allowing more jobs to run concurrently, etc.

CI/II EFFICIENCIES

- The adaptable string bound construct should be quoted whenever possible to give the compiler a clue as to the maximum length. This will often result in more efficient code being generated for adaptable strings.

- References to XDCL variables and variables declared within a SECTION will be made via the binding section end, consequently, an overhead is associated with the first reference.

- The code generator does not currently move invariant code out of loops. Consequently, access to variables through the binding section within a loop would be more efficient if the initial access to the variable is outside the loop.

- The reach of the load & store instructions on the Advanced System is limited to 2**16. When using large variables the offset may become greater than this threshold and result in an extra instruction being generated to handle the large offset. This would indicate organizing the more frequently used variables first in very large user stacks.

CM EFFICIENCIES

- Subranges should never be any larger then necessary. Having subranges larger than necessary negates CYBIL's range checking, and generally causes less efficient code. In particular, definitions of symbols such as int16 and uint16, which define 16-bit signed or 16-bit unsigned integer subranges should be avoided. These tend to cause register extend operations when used in arithmetic operations. For example, if 2 16-bit values are added, the values must be extended to 32 bits before adding them in order to guarantee a correct result.

CP EFFICIENCIES

- The UCSD p-system does not have an exclusive or instruction. Therefore, set references using the XOR operator generates a lot of code.

- Using long integer subranges results in less efficient code.

- The most commonly used variables should be entered first in the list of variables for a procedure. The first n variables in a procedure are accessed by a 1 byte instruction, the others by 2 bytes. Consequently

large structures (i.e. arrays) should be the last variables in the list.

- Using global variables in other modules should be avoided.
- Avoid FOR statements in favor of WHILE or REPEAT. They are faster and produce less code.
- Use base 0 for arrays rather than 1. E.g. use "ARRAY [0 .. n-1]" instead of "ARRAY [1 .. n]".

COMPILATION EFFICIENCIES

If compilation time is a factor the following items could be considered as they do affect the compilation rate.

- The generation of information to interface to the symbolic debuggers slows the compilation process.
- The generation of stylized code slows the compilation process.
- The generation of range checking code slows the compilation process.
- The selection of listings slows the compilation process. This includes the source listing, the cross reference listing and the attribute listing.
- Generating a source listing with the generated code included is slower than if just the source listing is being obtained.
- Actually, for the normal CYBIL user very little can be done to improve the compilation rate. However, rest assure that considerable effort has been expended to reduce the number of recompilations necessary to produce a debugged program.

IMPLEMENTATION LIMITATIONS

GENERAL

- Maximum number of lines in a single compilation unit is 65535.
- Maximum number of unique identifiers allowed in a single compilation unit is 16383.
- Maximum number of procedures in a single compilation unit is 999.
- Procedures can only be nested 255 levels deep.
- Maximum number of compile time variables used in conditional compilations is limited to 1023.
- Maximum number of error messages printed per module is 2000.
- Maximum number of elements defined in a single ordinal list is limited to 16384.
- Integer constants are restricted to 48 bits on the C170.

CC LIMITATIONS

- Case selector values limited to less than 2**17.
- Pointer fields within initialized packed records must be aligned for use within C170 capsules or overlay capsules.
- Packed arrays whose element size exceeds 2**17 bits gets a subscript range error.

CI/II LIMITATIONS

- Maximum number of lines in a single compilation unit is 32767 when run time error checking is selected.
- Nesting level of structured statements is limited to 63 levels deep.
- FOR statements can only be nested 15 levels deep.
- Procedures may only be nested 50 levels deep.
- Number of parameters passed to an xrefed procedure is 127, while an xrefed function is limited to 126.

- The reach of jump instructions is limited to 2**16 so the size compilation units should be appropriately controlled.
- The stack size of a single procedure is limited to 2**15 bytes.
- Long constants are not included in the debug symbol tables produced.

CM LIMITATIONS

- Maximum number of lines in a single compilation unit is 32767 when run time error checking is selected.
- Nesting level of structured statements is limited to 63 levels deep.
- FOR statements can only be nested 15 levels deep.
- Procedures may only be nested 50 levels deep.
- Number of parameters passed to an xrefed procedure is 127, while an xrefed function is limited to 126.
- The reach of jump instructions is limited to 2**16 so the size of a module should be appropriately controlled.
- The stack frame size is limited to 2**15 bytes.

CP LIMITATIONS

- In general the size of arrays and strings should be limited to less than 2**15 bytes.
- Maximum number of procedures in a single module is limited to 254.
- The maximum nesting level of procedures is 30.
- The use of long integer subranges is not allowed in the following areas:
  - Array subscripts,
  - As the <first char> or as the <substring length> on any string reference,
  - As the selector on a case statement,
  - As a actual parameter to a formal reference parameter of type integer.
  - As the control variable, starting value or ending value of a FOR statement.
- The result of a Stringrep operation on a floating point number is limited to 6 digits.

CS/SS LIMITATIONS

- Array size limited to 2**32.

COMPILER_AND_SPECIFICATION_DEVIATIONS

This section is intended to provide sufficient detail to be able to understand those features where the compiler implementation lags the language specification.

GENERAL

CYBIL_Implementation_-_Deviations

- #SEQ function.
- Restricting pointers to not point to data with less scope.
- Initialization of static pointers to NIL and zeroing the adaptable descriptor fields is not done.
- #SIZE of adaptable types.
- Run time checking on accessing fields of variant records not supported.
- #Current_stack_frame intrinsic.
- Support adaptable arrays of zero dimension.
- Double Precision Floating Point (LONGREAL).
- RESET TO with a relative pointer.
- STRLENGTH of constant of constant identifier.
- Library pragmat.

- Pre-defined identifiers are implemented as reserved words.

CC_DEVIATIONS

- Relative Pointer Types.
- General Intrinsics.
- Partial condition evaluation on OR operator not supported.
- Actual value parameters > 1 word must be addressable.

CI/II_DEVIATIONS

CM_DEVIATIONS

- Large value parameters are never copied.
- If a non-local exit from a function is done, the function result value is always undefined.
- Single Precision Floating Point (REAL).

CP_DEVIATIONS

- Static initialization.
- PUSH statement is not supported.
- Relative Pointers.

- General Intrinsics.
- Copies of adaptable value parameters are never made.

CS/SS_DEVIATIONS

- C200 Intrinsics.